An abstract Data Type for parallel Simulations based on Sparse Grids*

Michael May[1] and Thomas Schiekofer[2]

[1] Technical University of Munich, Institute of Computer Science,
Department of Computer Architecture and Computer Organization
Arcisstr.21,
D - 80290 Munich, Germany
email: maym@informatik.tu-muenchen.de
[2] University of Bonn, Institute of Applied Mathematics,
Department of Scientific Computing and Numerical Simulation,
Wegelerstr. 6,
D - 53115 Bonn, Germany
email: schiekof@iam.uni-bonn.de

Abstract. In this paper we introduce an abstract data type for the distributed representation and efficient handling of sparse grids on parallel architectures. The new data layout, implemented by means of the PVM message passing library, provides partitioning and dynamic load balancing transparent to the application making use of it. This enables parallelization of sequential partial differential equations (PDE) solvers based on sparse grids with hardly any source code modifications.

1 Sparse Grids

The numerical simulation of problems occuring in engineering and natural sciences need quite fine grids for discretization if they should supply reasonable results. The concept of adaptivity takes care of smoothness of the treated function, provides a grid which is appropriate to the given problem and allows efficient discretization. Adaptive full grids require only few grid points where the solution is smooth and possess many grid points where regularity vanishes.

Sparse grids were introduced 1990 by [7] (see also [2]) and represent a given function (e.g. a solution of a PDE) with even less grid points than adaptive full grids do. Moreover, sparse grids allow the concept of adaptivity, too.
Sparse grids only contain $\mathcal{O}(h_l^{-1} \cdot (\mathrm{ld}(h_l^{-1}))^{d-1})$ grid points compared to $\mathcal{O}(h_l^{-d})$ for usual full grids, where d denotes the dimension of the problem and h_l the mesh width on the boundary. On the other hand the interpolation error in the L_2 norm only increases slightly from order $\mathcal{O}(h_l^2)$ to $\mathcal{O}(h_l^2 \cdot (\mathrm{ld}(h_l^{-1}))^{d-1})$. Sparse grid points are a subset of grid points of a full grid. An important characteristic

* This work is being funded by the ministry for education, science, research and technology BMBF within the project PAR-CVD (Parallel simulation of Chemical Vapor Deposition processes)

of sparse grid points is that they can be described by a level and an index information (see equ. 1) which can be exploited for hashing.

The just mentioned properties of sparse grids are sufficient to understand the scope of this paper beginning in section 1.2. For the curious reader however, we give a short introduction into the theory of sparse grids in the next section. For more detailed information we refer to [7], [1], [2], [3] and [4].

1.1 Theoretical background

The typical *nodal basis* on $\Omega = [0,1]$ in one dimension is given by $B_l = \{\varphi_l^i(x) |\ \varphi_l^i(x_j) = \delta_{ij}, \varphi_l^i(x) \text{ piecewise linear}, 0 \le i,j \le 2^l\}$, where $x_j = j \cdot 2^{-l}$ $(0 \le j \le 2^l)$ denote equidistant grid points of grid Ω_l of mesh width 2^{-l} with $x_0 = 0$ and $x_{2^l} = 1$. Sparse grids are based on hierarchical bases H_l which can be obtained as follows:

$$H_1 = B_1$$
$$H_l = H_{l-1} \cup \{\varphi_l^j \in B^l | x_j \in \Omega_l / \Omega_{l-1}\}$$

Figure 1 shows the hierarchical basis in the $1D$-case for $l = 3$. It can easily be verified that $span\{B_l\} = span\{H_l\}$ $(l \in \mathbb{N})$.

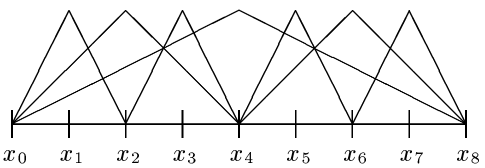

Figure 1: hierarchical basis in $1D$ for $l = 3$
(basis functions on the boundary are neglected)

The basis functions for the d-dimensional nodal basis on the hypercube $[0,1]^d$ can be defined as the tensor product basis functions of the one-dimensional nodal basis functions, that is

$$\varphi_{l_1,\ldots,l_d}^{i_1,\ldots,i_d}(x^{(1)},\ldots,x^{(d)}) = \varphi_{l_1}^{i_1}(x^{(1)}) \cdot \ldots \cdot \varphi_{l_d}^{i_d}(x^{(d)}). \tag{1}$$

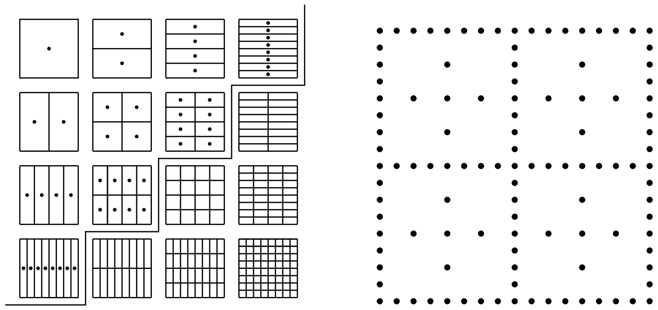

Figure 2: The grids Ω_{l_1,l_2} in the case $d = 2$ for $1 \le l_1, l_2 \le 4$ and sparse grid $S_{4,4}^{(2)}$

We introduce the grids $\Omega_{l_1,\ldots,l_d}$ on $[0,1]^d$ with mesh width 2^{l_k} $(1 \le k \le d)$ belonging to coordinate direction $x^{(k)}$. The space $S^{(d)}_{l_1,\ldots,l_d}$ denotes the space of piecewise d-linear functions on grid points of grid $\Omega_{l_1,\ldots,l_d}$. Additionally, we need spaces $T_{i_1,\ldots,i_d}$ which consist of piecewise d-linear functions on $\Omega_{i_1,\ldots,i_d}$ vanishing on all grid points of grids $\Omega_{k_1,\ldots,k_d}$ with $k_j \le i_j$ for $1 \le j \le d$ and $k_1 + \ldots + k_d < i_1 + \ldots + i_d$. Figure 2 shows the grids Ω_{i_1,i_2} induced by T_{i_1,i_2} in the $2D$-case for $1 \le i_1, i_2 \le 4$ (grid points on the boundary are neglected).

Using $T_{i_1,\ldots,i_d}$ we can rewrite $S^{(d)}_{l_1,\ldots,l_d} = \oplus_{i_1=1}^{l_1} \cdots \oplus_{i_d=1}^{l_d} T_{i_1,\ldots,i_d}$. The interpolation of a d-dimensional continuous function $u(x^{(1)}, \ldots, x^{(d)})$ yields

$$u_I^{(d)}(x^{(1)},\ldots,x^{(d)}) = \sum_{i_1=1}^{l_1} \cdots \sum_{i_d=1}^{l_d} u_{i_1,\ldots,i_d} \quad \text{with}$$

$$u_{i_1,\ldots,i_d} = \sum_{k_1=1}^{2^{i_1}-1} \cdots \sum_{k_d=1}^{2^{i_d}-1} c^{i_1,\ldots,i_d}_{k_1,\ldots,k_d} \varphi^{i_1,\ldots,i_d}_{k_1,\ldots,k_d}$$

with the hierarchical surplusses $c^{i_1,\ldots,i_d}_{l_1,\ldots,l_d}$ (cf. [1]). In the $1D$-case the hierarchical surplus of a node x_k is $c^i_k = u(x_k) - 0.5 \cdot (u(x_{h_l}(x_k)) + u(x_{h_r}(x_k)))$ where $x_{h_l}(x_k)$ and $x_{h_r}(x_k)$ are the left and right hierarchical neighbor of x_k (for example in figure 1 x_0 and x_4 are the hierarchical neighbors of x_2) and $u(\cdot)$ is the function which is to be represented. The computation of a d-dimensional hierarchical surplus is straightforward using the tensor product notation.

Let us now consider the $2D$-case for the introduction of sparse grids. The error contribution of a single space T_{i_1,i_2} with respect to the L_2-norm is of order $\mathcal{O}(4^{-i_1-i_2})$. Obviously, all spaces T_{i_1,i_2} with $i_1 + i_2 = c$ (c fixed) are of same dimension and give the same order of contribution to the error. Using a triangular scheme of T_{i_1,i_2} (as indicated in figure 2 by a line) instead of the quadratic scheme used by full grids yields the *sparse grids*.

Let us return to the d-dimensional case and denote the space of piecewise d-linear function on a sparse grid by $\tilde{S}^{(d)}_{l,\ldots,l}$. Using $T_{i_1,\ldots,i_d}$ again we can write $\tilde{S}^{(d)}_{l,\ldots,l} = \oplus_{i_1+\ldots i_d \le l+d-1} T_{i_1,\ldots,i_d}$. The interpolation of a continuous function $u(\cdot)$ in the sparse grid space yields now

$$\tilde{u}^{(d)}_{l,I} = \sum_{i_1+\ldots+i_d \le l+d-1} u_{i_1,\ldots,i_d}.$$

Evidently, sparse grids need much less grid points than full grids. They only involve $\mathcal{O}(h_l^{-1} \cdot (\mathrm{ld}(h_l^{-1}))^{d-1})$ grid points compared to $\mathcal{O}(h_l^{-d})$ for full grids where h_l denotes the mesh width on the boundary. error in the L_2-norm is only slightly worse than in the full grid case: it is of order $\mathcal{O}(h_l^2 \cdot (\mathrm{ld}(h_l^{-1}))^{d-1})$ instead of $\mathcal{O}(h_l^2)$. The same behavior can be obtained in the L_∞-norm. If we measure the error contribution of a space $T_{i_1,\ldots,i_d}$ with respect to the energy norm we can eliminate additional spaces $T_{i_1,\ldots,i_d}$ of our triangular scheme which was deduced by using the L_2-norm. In this case we get energy based sparse grids which consists only of $\mathcal{O}(h_l^{-1})$ grid points, but have same error order $\mathcal{O}(h_l)$ as the full grid.

1.2 Data Structures

In literature the only data structure considered useful to implement sparse grids are systems of binary trees. They represent the hierarchical structure and the low density of sparse grids very well, in contrast to inefficient and costly standard arrays. The interpretation of the individual binary trees is, for instance in a three-dimensional case, as follows:

A first binary tree, whose nodes represent *grid planes*, contains second order binary trees with nodes representing *grid lines*. On each of these second order nodes, third order binary trees eventually represent the actual *grid points* along the given lines. The problems arising from this data structure layout are obvious: Depending on the target function, the binary trees can become arbitrarily deep and hence degenerate into linked lists. This fact hinders both the efficient access and distribution of the data onto the available processors, which are only possible from balanced trees.

For use with sparse grid based PDE solvers we thus need efficient implementations of search operations for arbitrary grid points, while insertions and deletions during the building and destruction of the sparse grid may be slow.

Additionally pointered Binary Trees As a first step to avoid the problem of accessing neighboring points in the sparse grid, one could think of additional pointers to represent nearest neighbors and hierarchical son and father relationship. Still efficient access to the grid points in a possibly degenerated binary tree cannot be guaranteed.

Hash Tables A way to accelerate the access is to compute the address of the data instead of searching for it. This notion leads to hashing the grid points into a Hashing Table and resolving collisions in linked lists. In this layout no dimension of the grid is preferred and all access to neighboring grid points is equally fast and easy. However, the distribution of the Hashing Table onto the processors is hindered by the arbitrary scattering of data over the available table space, destroying the needed data *convexity*, i.e. adjacent data is *not* mapped on a neighboring space in the hash table.

Partitioned Hash Tables In order to avoid the scattering of the grid points, one could partition the sparse grid into convex areas, i.e. d-dimensional quads, and give each quad its own hash table, which then could be mapped onto the available processors. However, the partitioning of the hash tables is static, so any adapting of the sparse grid to the target function will lead to in-equal load of the processors.

In order to handle these limitations we need a representation of sparse grids that offers both efficient access and easy data partitioning for parallel processing. In the next section we present a module implementing an abstract data type `Sparse_Grid` for the parallel representation of such grids satisfying the demands just mentioned.

2 The distributed abstract Data Type `Sparse_Grid`

An abstract data type offers its functionality as a black box, i.e. its implementation and properties (variables) are encapsulated inside the module and can only be accessed by means of the exported module methods. This way the numerical algorithm using the data type `Sparse_Grid` does not and need not know anything about the internal representation and distribution of the grid points. Modifications to step from a sequential numerical program to a parallel one are thus minimal.

An incarnation of the data type is characterized by the size of the individual dimensions and the number of maximal grid points leading to the notion of a *virtual* array as the sparse grid points are internally stored in hash tables instead of arrays. Instances of `Sparse_Grid` can be initialized and destroyed by the methods

`SG_alloc()`, `SG_dealloc()`

where the first method needs the maximal number of grid points and the grid sizes as parameters.

The properties of each individual grid point are the level and index information (see equ. 1) and a special field to hold additional information like function values or the hierarchical surplus (see 1.1). In order to read and set these properties there are the methods

`SG_insertnode()`, `SG_deletenode`, `SG_findnode()`, `SG_set()`

which all need the following parameters: a sparse grid `SG_grid`, the level and the index of the grid point and the additional information. These parameters do not have all to be set, depending whether the access is read or write.

In order to initialize, synchronize and re-gather the data distribution and parallel calculation there are three more external methods,

`SG_distribute()`, `SG_sync()`, `SG_collect()`,

which need an instance of `Sparse_Grid` and the number of processors as parameters.

Everything else, for instance the partitioning and load balancing schemes by means of processor *splits* and processor *merges* (see section 2.2), the internals of synchronization, the storing and accessing of grid points in processor local hash tables, is fully transparent to the numerical algorithm.

The data structures inside the sparse grid module are based on a multikey file structure for accessing disk storage presented by [6]. In the subsequent subsections we will give an overview of all hidden features of the abstract data type `Sparse_Grid` and discuss its suitability for partitioning and load balancing.

2.1 Process Arrays

For notational simplicity we only regard the two-dimensional case (d=2), as the adaption to the general case ($d > 2$) is straight-forward.

For a parallel computation the target area covered by a sparse grid should be divided into subregions of uniform point density. This is achieved by means of a so called *process array*, a two-dimensional array

`process_array[0..nx-1][0..ny-1]`,

defining the correspondence of convex 2-dimensional quads of the grid to associated processors or processes. Each cell of `process_array` holds the identification number of the processor or process responsible for the appropriate subregion. The grid points in these subregions are stored in (local) processor memory in (individual) hash tables.

The partition of sparse grid (that yields the process array) is determined by so called *scales*, vectors

`xscale[0..nx]` *and* `yscale[0..ny]`

describing the partition of each dimension of the grid respectively.

These data structures are replicated on each processor of the parallel machine.

The access (find, insert, delete) to a sparse grid point with level l and index i is as follows (see figure 3):

1. The level and index of the grid point are checked against the x-scale and y-scale to determine the appropriate row x and column y in the process array.
2. Next it is checked whether the pointer `process_array[x][y]` matches the processor of the computing process. If so, the grid point is accessed in the local processor memory by efficient hashing techniques. A write access can lead to subsequent processor splits or merges[3].
3. If `process_array[x][y]` does not match, the access to the grid point is delegated by passing a message to the corresponding processor.

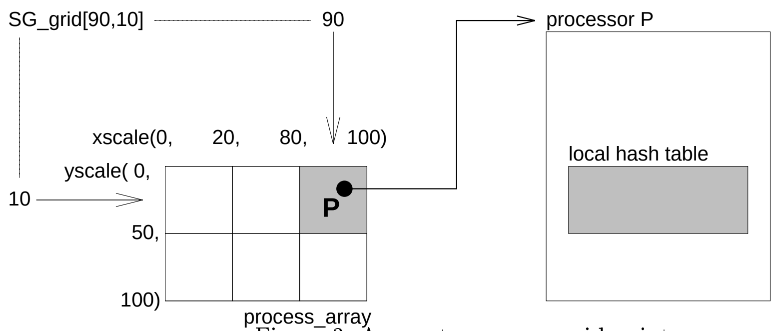

Figure 3: Access to a sparse grid point

[3] which triggers a broadcast update of the process_array and the scales to the remaining processors

2.2 Splitting and Merging

Partitioning and dynamic load balancing are achieved by means of processor *splits* and processor *merges*, if individual processor loads, i.e. the number of locally stored sparse grid points, reach a maximal (minimal) value by the repeated insertion (deletion) of grid points. Two parameters influence the dynamic correspondence of the processors to the sparse grid subregions, the *splitting policy* and the *merging policy*.

1. The **splitting policy** is characterized by three features:
   - □ The (maximal) *processor load*, of which any overflow leads to a split.
   - □ The *dimension* that is chosen for the next split, i.e. the axis to which the partitioning hyper-plane is orthogonal. For instance, a very simple splitting policy would choose the dimensions cyclically. More elaborate policies could favor some dimensions and thus lead to higher resolutions.
   - □ The *location* of the split point, i.e. the point at which the linear scale is partitioned
2. The **merging policy** is determined by the following decisions:
   - □ The (minimal) *processor load*, of which any underflow triggers a merge operation.
   - □ Which pair of adjacent subregions is a possible *candidate* for a merge. Examples for algorithms to choose possible merge partners are the well known buddy system and the more general neighbor system.
   - □ If there are several candidates, which one wins the *competition*.

These policies of the abstract data type enable an equal distribution of the virtual sparse grid array over the processors and re-balance the processor load after each split or merge implementing a dynamic partitioning.

As there is no a-priori information about the number of points contained in a sparse grid, a dynamic distribution using subsequent processor splits from the very beginning can lead to unequally balanced grid points and thus processor loads. In such cases it is more efficient to build up the full sparse grid on all processors, split the grid in subregions and then delete the grid points not needed for the local computation on each processor respectively. A further adaption of the sparse grid to the target function can be balanced again by processor splits.

2.3 Hash Tables

For memory and performance reasons the points of the virtual sparse grid are internally stored in hash tables. In the case of sparse grids it is evident from the introduction to sparse grids (equ. 1) that information of level and index is sufficient to identify a sparse grid point. Consequently, the position of a node is determined by the result of a hash function that computes an array-index of the hash table from the parameters level and index.

2.4 Synchronization with PVM

As already mentioned above, access to non-local grid points is delegated by passing a message to the processor responsible for the appropriate subregion. PVM messages in the abstract data type `Sparse_Grid` are used in a uniform manner:

1. The messages are tagged with a flag indicating the type of action to be performed by the receiving processor, and
2. contain all the data necessary for the indicated operation.

For example, in order to insert a new grid point in the grid region of another processor, a message containing the level, index and additional information is packed, tagged with `SIG_INS` and sent to the appropriate processor. At a given time, the appropriate function on the remote processor, here `SG_insertnode()`, is called with the parameters that were delivered in the message.
The message tags are

`SIG_INS`, `SIG_DEL`, `SIG_GET`, `SIG_SPLIT`, `SIG_MERGE`

to signal a nonlocal insertion, deletion, find, split and merge operation respectively.

The messages are handled and buffered in a message queue by the PVM daemon until they are received by the appropriate PVM process. This message queue is worked off by a message handler inside the module, `SG_sync()`. Only by calling `SG_sync()` messages will be received and corresponding actions performed. By placing calls to the `SG` message handler on synchronization points in the numerical application, the user has complete control over the strictness and timing of synchronization among the processors.
For instance, in conventional scientific computing, a call to `SG_sync()` should be placed after every time step, i.e. at the end of the time-step loop. To support local time stepping inside the target domains, `SG_sync()` could be called only every, say 50 time steps. Eventually, calling `SG_sync()` after every write access to the sparse grid would implement a very strict synchronization.

3 Conclusion and Future Work

We have presented an abstract data type built on top of PVM to distribute and parallelize sparse grids nearly transparently to the numerical application. As a case study we are about to parallelize a sparse grid based Navier-Stokes-solver for use with `Fastest 3D`, a simulation package for chemical vapor deposition processes in the electronic and optoelectronic field.
As the data type `Sparse_Grid` is suitable for grids of arbitrary dimensions, it is obvious to implement simulations of d-dimensional problems as well i.e. to handle heat transport and chemical reaction equations.

References

1. Bungartz, H.-J.: *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung*,
   Ph.D. Thesis, Technical University of Munich (1992)
2. Griebel, M.: *A parallelizable and vectorizable multi-level algorithm on sparse grids*, in Parallel Algorithms for partial differential equations, Notes on Numerical Fluid Mechanics, Volume 31, pp. 94-100, W. Hackbusch, ed., Vieweg, Braunschweig, 1991
3. Griebel, M.: *Parallel Multigrid Methods on Sparse Grids*,
   in Multigrid Methods III, Proceedings of the 3rd European Conference on Multigrid Methods, Bonn, Oktober 1990, International Series of Numerical Mathematics (Vol. 98), W. Hackbusch and U. Trottenberg, ed., Birkhäuser, Basel, 1991 (also: Technical University of Munich, SFB report no. 342/30/90 A)
4. Griebel, M.: *The combination technique for the sparse grid solution of PDE's on multiprocessor machines*,
   Technical University of Munich, SFB report no. 342/14/91 A, 1991
5. Griebel M., Schneider M. and Zenger C. *A combination technique for the solution of sparse grid problems*,
   in Iterative Methods in Linear Algebra, pp. 263-281, IMACS, P. de Groen und R. Beauwens, ed., Elsevier, North Holland, 1992, (also: Technical University of Munich, SFB report no. 342/19/90 A)
6. Nievergelt, J., Hinterberger, H., Sevcik, K. C.: *The Grid File: An Adaptable, Symmetric Multikey File Structure*,
   ACM Transactions on Database Systems, Vol. 9, No. 1 (1981)
7. Zenger, C.: *Sparse Grids in Parallel Algorithms for Partial Differential Equations*, Proceedings of the Sixth GAMM-Seminar, Kiel, 19.01.-21.01.1990, Notes on Numerical Fluid Mechanics Vol. 31, W. Hackbusch, ed., Vieweg, Braunschweig, 1991 (also: Technical University of Munich, SFB report no. 342/18/90 A)
8. Zenger, C.: *Hierarchische Datenstrukturen für glatte Funktionen mehrerer Veränderlicher*,
   Informatik und Mathematik, Broy, M., Springer, Berlin (1991)

This article was processed using the LaTeX macro package with LLNCS style